COMPUTER SCIENCE A

SECTION II

Time—1 hour and 45 minutes

Number of questions—4

Percent of total grade—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN JAVA.

Notes:

- Assume that the classes listed in the Java Quick Reference have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not `null` and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

1. Consider the following partial declaration for a `WordScrambler` class. The constructor for the `WordScrambler` class takes an even-length array of `String` objects and initializes the instance variable `scrambledWords`.

```
public class WordScrambler
{
  private String[] scrambledWords;

  /** @param wordArr an array of String objects
   *          Precondition: wordArr.length is even
   */
  public WordScrambler(String[] wordArr)
  {
    scrambledWords = mixedWords(wordArr);
  }


  /** @param word1 a String of characters
   *  @param word2 a String of characters
   *  @return a String that contains the first half of word1 and the second half of word2
   */
  private String recombine(String word1, String word2)
  {  /* to be implemented in part (a) */  }


  /** @param words an array of String objects
   *          Precondition: words.length is even
   *  @return an array of String objects created by recombining pairs of strings in array words
   *  Postcondition: the length of the returned array is words.length
   */
  private String[] mixedWords(String[] words)
  {  /* to be implemented in part (b) */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

GO ON TO THE NEXT PAGE.

(a) Write the `WordScrambler` method `recombine`. This method returns a `String` created from its two `String` parameters as follows.

- take the first half of `word1`
- take the second half of `word2`
- concatenate the two halves and return the new string.

For example, the following table shows some results of calling `recombine`. Note that if a word has an odd number of letters, the second half of the word contains the extra letter.

`word1`	`word2`	`recombine(word1, word2)`
`"apple"`	`"pear"`	`"apar"`
`"pear"`	`"apple"`	`"peple"`

Complete method `recombine` below.

```
/** @param word1 a String of characters
 *  @param word2 a String of characters
 *  @return a String that contains the first half of word1 and the second half of word2
 */
private String recombine(String word1, String word2)
```

(b) Write the `WordScrambler` method `mixedWords`. This method creates and returns a new array of `String` objects as follows.

It takes the first pair of strings in `words` and combines them to produce a pair of strings to be included in the array returned by the method. If this pair of strings consists of `w1` and `w2`, the method should include the result of calling `recombine` with `w1` and `w2` as arguments and should also include the result of calling `recombine` with `w2` and `w1` as arguments. The next two strings, if they exist, would form the next pair to be processed by this method. The method should continue until all the strings in `words` have been processed in this way and the new array has been filled. For example, if the array `words` contains the following elements:

```
{"apple", "pear", "this", "cat"}
```

then the call `mixedWords(words)` should return the following array.

```
{"apar", "peple", "that", "cis"}
```

In writing `mixedWords`, you may call `recombine`. Assume that `recombine` works as specified, regardless of what you wrote in part (a).

Complete method `mixedWords` below.

```
/** @param words an array of String objects
 *          Precondition: words.length is even
 *  @return an array of String objects created by recombining pairs of strings in array words
 *  Postcondition: the length of the returned array is words.length
 */
private String[] mixedWords(String[] words)
```

GO ON TO THE NEXT PAGE.

2. An array of positive integer values has the *mountain* property if the elements are ordered such that successive values increase until a maximum value (the peak of the mountain) is reached and then the successive values decrease. The `Mountain` class declaration shown below contains methods that can be used to determine if an array has the mountain property. You will implement two methods in the `Mountain` class.

```
public class Mountain
{
  /** @param array an array of positive integer values
   *  @param stop the last index to check
   *          Precondition: 0 ≤ stop < array.length
   *  @return true if for each j such that 0 ≤ j < stop, array[j] < array[j + 1] ;
   *          false otherwise
   */
  public static boolean isIncreasing(int[] array, int stop)
  {  /*  implementation not shown  */  }


  /** @param array an array of positive integer values
   *  @param start the first index to check
   *        Precondition: 0 ≤ start < array.length - 1
   *  @return true if for each j such that start ≤ j < array.length - 1,
   *                array[j] > array[j + 1];
   *          false otherwise
   */
  public static boolean isDecreasing(int[] array, int start)
  {  /*  implementation not shown  */  }


  /** @param array an array of positive integer values
   *          Precondition: array.length > 0
   *  @return the index of the first peak (local maximum) in the array, if it exists;
   *          -1 otherwise
   */
  public static int getPeakIndex(int[] array)
  {  /*  to be implemented in part (a)  */  }


  /** @param array an array of positive integer values
   *          Precondition: array.length > 0
   *  @return true if array contains values ordered as a mountain;
   *          false otherwise
   */
  public static boolean isMountain(int[] array)
  {  /*  to be implemented in part (b)  */  }

  // There may be instance variables, constructors, and methods that are not shown.

}
```

GO ON TO THE NEXT PAGE.

(a) Write the `Mountain` method `getPeakIndex`. Method `getPeakIndex` returns the index of the first peak found in the parameter `array`, if one exists. A peak is defined as an element whose value is greater than the value of the element immediately before it and is also greater than the value of the element immediately after it. Method `getPeakIndex` starts at the beginning of the array and returns the index of the first peak that is found or -1 if no peak is found.

For example, the following table illustrates the results of several calls to `getPeakIndex`.

arr	getPeakIndex(arr)
{11, 22, 33, 22, 11}	2
{11, 22, 11, 22, 11}	1
{11, 22, 33, 55, 77}	-1
{99, 33, 55, 77, 120}	-1
{99, 33, 55, 77, 55}	3
{33, 22, 11}	-1

Complete method `getPeakIndex` below.

```
/** @param array an array of positive integer values
 *          Precondition: array.length > 0
 *  @return the index of the first peak (local maximum) in the array, if it exists;
 *           -1 otherwise
 */
public static int getPeakIndex(int[] array)
```

(b) Write the `Mountain` method `isMountain`. Method `isMountain` returns `true` if the values in the parameter `array` are ordered as a mountain; otherwise, it returns `false`. The values in `array` are ordered as a mountain if all three of the following conditions hold.

- There must be a peak.
- The array elements with an index smaller than the peak's index must appear in increasing order.
- The array elements with an index larger than the peak's index must appear in decreasing order.

For example, the following table illustrates the results of several calls to `isMountain`.

arr	isMountain(arr)
{1, 2, 3, 2, 1}	true
{1, 2, 1, 2, 1}	false
{1, 2, 3, 1, 5}	false
{1, 4, 2, 1, 0}	true
{9, 3, 5, 7, 5}	false
{3, 2, 1}	false

In writing `isMountain`, assume that `getPeakIndex` works as specified, regardless of what you wrote in part (a).

Complete method `isMountain` below.

```
/** @param array an array of positive integer values
 *         Precondition: array.length > 0
 *  @return true if array contains values ordered as a mountain;
 *          false otherwise
 */
public static boolean isMountain(int[] array)
```